System Programming Project 4

담당 교수 : 박성용 교수님

이름 : 김규빈

학번 : 202020152

1. **개발 목표**

이 프로젝트는 implicit free list와 segregated free list 및 best fit을 사용하여 dynamic memory allocator를 구현한다. block은 해제 시 즉시 병합되고, 할당된 block은 남은 부분이 유효한 free block이 될 만큼 충분히 큰 경우 분할된다. 위 구현은 메모리 효율성과 할당 속도 간의 균형을 맞추는 것을 목표로 한다.

2. **로직**

- 초기화
- 1. free list 초기화:
- • initialize_free_list 함수를 호출하여 segregated free list 배열을 초기화한다.
- • segregated_free_lists 배열을 할당하고 각 요소를 NULL로 설정하여 빈 상태로 만든다.
- 2. 프롤로그 block 초기화:
- • initialize_prologue_block 함수를 호출하여 힙의 프롤로그 block을 설정한다.
- • prologue_block을 할당하고, 프롤로그 header, footer 및 에필로그 header를 설정한다.
- 3.힙 확장:
- • mm_init 함수에서 extend_heap 함수를 호출하여 초기 힙을 확장한다.
- • 초기 힙 확장이 실패하면 초기화 함수는 -1을 반환하여 오류를 나타낸다.

- 메모리 할당
- 1. 요청된 크기 조정:
- • mm_malloc 함수에서 요청된 크기를 조정한다.
- • 크기가 DWORD_SIZE 이하인 경우 최소 크기인 2 * DWORD_SIZE로 설정한다.
- • 그렇지 않으면 DWORD_SIZE의 배수로 올림한다.

- 2. 적절한 block 찾기:
- • find_fit 함수를 호출하여 적절한 free block을 찾는다.
- • find_list_index 함수는 크기에 따라 적절한 list 인덱스를 결정한다.
- • free list를 순회하며 최적의 block을 찾는다. best fit 알고리즘을 사용하여 가장 작은 차이의 block을 선택한다.
- 3. block 배치:
- • place 함수를 호출하여 요청된 크기의 block을 free block에 배치한다.
- • block이 충분히 크면 분할하여 남은 부분을 free list에 삽입한다.
- 4. 힙 확장:
- • 적절한 block을 찾지 못한 경우 extend_heap 함수를 호출하여 힙을 확장한다.
- • 확장된 block을 병합하여 free list에 삽입하고, 다시 배치한다.

- 메모리 해제
- 1. block 해제:
- • mm_free 함수에서 block을 해제한다.
- • header와 footer를 업데이트하여 block을 free 상태로 표시한다.
- • block을 free list에 삽입한다.
- 2. block 병합:
- • coalesce 함수를 호출하여 인접한 free block을 병합한다.
- • 이전 block과 다음 block의 할당 상태를 확인하여 필요한 경우 병합한다.
- • 병합된 block을 free list에 다시 삽입한다.

- block 병합
- 1. 다음 block과 병합:

- • coalesce_with_next 함수를 호출하여 현재 block과 다음 block을 병합한다.
- • 다음 block을 free list에서 삭제하고, 병합된 block의 크기를 업데이트하여 header와 footer를 수정한다.
- 2. 이전 block과 병합:
- • coalesce_with_prev 함수를 호출하여 현재 block과 이전 block을 병합한다.
- • 이전 block을 free list에서 삭제하고, 병합된 block의 크기를 업데이트하여 header와 footer를 수정한다.

- block 삽입 및 삭제
- 1. free block 삽입:
- • insert_free_block 함수를 호출하여 free block을 segregated free list에 삽입한다.
- • block의 크기에 따라 적절한 인덱스를 찾아 list에 삽입한다.
- • list를 순회하며 적절한 위치를 찾아 block을 삽입한다.
- 2. free block 삭제:
- • delete_free_block 함수를 호출하여 free block을 segregated free list에서 삭제한다.
- • block의 위치에 따라 포인터를 업데이트하여 list에서 제거한다.

- 메모리 재할당
- 1. block 재할당:
- • mm_realloc 함수에서 block을 재할당한다.
- • 요청된 크기를 조정하고, 현재 block이 충분히 큰 경우 그대로 반환한다.
- • 다음 block이 free block이고 충분히 큰 경우 병합하여 재할당한다.
- • 새로운 block을 할당하고, 기존 데이터를 복사한 후 이전 block을 해제한다.

3. **매크로 설명**

기본 상수

- ALIGNMENT: 정렬 크기, 8바이트로 설정
- WORD_SIZE: 워드 크기, 4바이트로 설정
- DWORD_SIZE: 더블 워드 크기, 8바이트로 설정

Chunk 크기

- INITIAL_CHUNK_SIZE: 힙 확장을 위한 초기 chunk 크기, 4096바이트로 설정
- MIN_CHUNK_SIZE: 작은 할당을 위한 최소 chunk 크기, 64바이트로 설정.

유틸리티

- MAX_VALUE(x, y): x와 y 중 큰 값을 반환.
- MIN_VALUE(x, y): x와 y 중 작은 값을 반환.
- PACK_SIZE_AND_ALLOC(size, alloc): 크기와 할당 비트를 하나의 워드로 패킹.
- WRITE_WORD(p, val): 주소 p에 워드 val을 씀.
- READ_SIZE(p): 주소 p에서 크기 필드를 읽음.
- READ_ALLOC(p): 주소 p에서 할당 필드를 읽음.
- READ_WORD(p): 주소 p에서 워드를 읽음.
- READ_TAG(p): 주소 p에서 태그 비트를 읽음.

Block 연산

- GET_HEADER(bp): block 포인터 bp의 header 주소 계산.
- GET_FOOTER(bp): block 포인터 bp의 footer 주소 계산.
- GET_NEXT_BLOCK(bp): 힙에서 다음 block의 주소 계산.
- GET_PREV_BLOCK(bp): 힙에서 이전 block의 주소 계산.

Free list 연산

- GET_NEXT_FREE_PTR(ptr): free block에서 다음 free pointer를 얻음.

- GET_NEXT_FREE_REF(ptr): 다음 free block의 참조를 얻음.
- GET_PREV_FREE_PTR(ptr): free block에서 이전 free pointer를 얻음.
- GET_PREV_FREE_REF(ptr): 이전 free block의 참조를 얻음.

기타

- NUM_FREE_LISTS: segregated free list의 수, 20으로 설정.
- ALIGN_SIZE(size): size를 ALIGNMENT의 배수로 올림.

4. 코드

- 초기화

- ```
  static void initialize_free_list(void);
  static void initialize_prologue_block(void);
  int mm_init(void);
  ```

- ```
  /* free 목록을 초기화 */
  static void initialize_free_list(void)
  {
      segregated_free_lists = (void **)malloc(sizeof(void *) * NUM_FREE_LISTS); // free 목록을 위한 메모리 할당
      if (segregated_free_lists == NULL)
      {
          fprintf(stderr, "Error: Could not initialize free list.\n"); // 오류 메시지 출력
          exit(EXIT_FAILURE);                                          // 프로그램 종료
      }
      for (int index = 0; index < NUM_FREE_LISTS; index++)
      {
          segregated_free_lists[index] = NULL; // 각 목록을 NULL로 초기화
      }
  }
  ```

  - initialize_free_list: segregated free list를 초기화한다.
  - segregated_free_lists 배열을 할당하고, 각 list를 NULL로 초기화하여 빈 상태로 만든다. 만약 메모리 할당에 실패하면 프로그램을 종료한다.

- ```
  /* Prologue block을 설정 */
  static void initialize_prologue_block(void)
  {
      if ((long)(prologue_block = mem_sbrk(4 * WORD_SIZE)) == -1) // 메모리 할당 요청
      {
          fprintf(stderr, "Error: Could not set prologue.\n"); // 오류 메시지 출력
          exit(EXIT_FAILURE);                                  // 프로그램 종료
      }
      WRITE_WORD(prologue_block, 0);                                                  // 패딩
      WRITE_WORD(prologue_block + 1 * WORD_SIZE, PACK_SIZE_AND_ALLOC(DWORD_SIZE, 1)); // Prologue header
      WRITE_WORD(prologue_block + 2 * WORD_SIZE, PACK_SIZE_AND_ALLOC(DWORD_SIZE, 1)); // Prologue footer
      WRITE_WORD(prologue_block + 3 * WORD_SIZE, PACK_SIZE_AND_ALLOC(0, 1));          // Epilogue header
  }
  ```
  ```

- initialize_prologue_block: 프롤로그 및 에필로그 block을 설정한다.
- prologue_block을 할당하고, 프롤로그 header, footer 및 에필로그 header를 설정한다. 프롤로그 block은 할당된 block으로 간주되고, 에필로그는 할당되지 않은 block의 끝을 나타낸다.

```
int mm_init(void)
{
    initialize_free_list();                  // free 목록 초기화
    initialize_prologue_block();             // Prologue block 설정
    if (extend_heap(MIN_CHUNK_SIZE) == NULL) // heap 확장
    {
        return -1; // 오류 발생 시 -1 반환
    }
    return 0; // 성공 시 0 반환
}
```

- mm_init: 메모리 관리자를 초기화한다.
- initialize_free_list와 initialize_prologue_block을 호출하여 초기화하고, 초기 heap을 확장한다. heap 확장에 실패하면 -1을 반환하여 초기화 실패를 나타낸다.

- mm_init에서 initialize_free_list와 initialize_prologue_block을 호출하여 초기화한다. free_lists는 segregated list를 관리하기 위해 사용하는 변수이다. free_lists는 연결 list 배열로 각 인덱스는 해당 크기의 free block을 연결한다.
- Block 삽입 및 삭제

```
/* block 삽입 및 삭제 */
static void insert_free_block(void *ptr, size_t size); // free block을 목록에 삽입하는 함수
static void delete_free_block(void *ptr);              // free block을 목록에서 삭제하는 함수
static int find_list_index(size_t size);               // 크기에 맞는 목록 인덱스를 찾는 함수
```

```c
/* free block을 목록에 삽입 */
static void insert_free_block(void *ptr, size_t size)
{
    int index = find_list_index(size);         // 크기에 맞는 목록 인덱스 찾기
    void *next = segregated_free_lists[index]; // 다음 block
    void *prev = NULL;                         // 이전 block

    // free 목록을 순회하며 적절한 위치 찾기
    while (next != NULL && size > READ_SIZE(GET_HEADER(next)))
    {
        prev = next;                    // 이전 block 업데이트
        next = GET_NEXT_FREE_REF(next); // 다음 block으로 이동
    }

    // block 삽입
    if (next != NULL)
    {
        if (prev != NULL)
        {
            WRITE_WORD(GET_NEXT_FREE_PTR(prev), (unsigned int)ptr); // 이전 block의 다음 포인터 설정
            WRITE_WORD(GET_PREV_FREE_PTR(next), (unsigned int)ptr); // 다음 block의 이전 포인터 설정
            WRITE_WORD(GET_NEXT_FREE_PTR(ptr), (unsigned int)next); // 현재 block의 다음 포인터 설정
            WRITE_WORD(GET_PREV_FREE_PTR(ptr), (unsigned int)prev); // 현재 block의 이전 포인터 설정
        }
        else
        {
            WRITE_WORD(GET_PREV_FREE_PTR(next), (unsigned int)ptr); // 다음 block의 이전 포인터 설정
            WRITE_WORD(GET_NEXT_FREE_PTR(ptr), (unsigned int)next); // 현재 block의 다음 포인터 설정
            WRITE_WORD(GET_PREV_FREE_PTR(ptr), 0);                  // 현재 block의 이전 포인터 설정
            segregated_free_lists[index] = ptr;                     // free 목록의 인덱스 설정
        }
    }
    else
    {
        if (prev != NULL)
        {
            WRITE_WORD(GET_NEXT_FREE_PTR(prev), (unsigned int)ptr); // 이전 block의 다음 포인터 설정
            WRITE_WORD(GET_PREV_FREE_PTR(ptr), (unsigned int)prev); // 현재 block의 이전 포인터 설정
            WRITE_WORD(GET_NEXT_FREE_PTR(ptr), 0);                  // 현재 block의 다음 포인터 설정
        }
        else
        {
            WRITE_WORD(GET_NEXT_FREE_PTR(ptr), 0); // 현재 block의 다음 포인터 설정
            WRITE_WORD(GET_PREV_FREE_PTR(ptr), 0); // 현재 block의 이전 포인터 설정
            segregated_free_lists[index] = ptr;    // free 목록의 인덱스 설정
        }
    }
}
```

- ■ insert_free_block: free block을 segregated free list에 삽입한다.
- ■ 새로운 free block을 segregated list에 삽입하는 기능을 하다. block의 크기에 따라 적절한 인덱스를 찾고, list를 순회하여 적절한 위치에 block을 삽입한다.

-
```c
/* free block을 목록에서 삭제 */
static void delete_free_block(void *ptr)
{
    int index = find_list_index(READ_SIZE(GET_HEADER(ptr))); // 크기에 맞는 목록 인덱스 찾기
    void *next = (void *)READ_WORD(GET_NEXT_FREE_PTR(ptr));  // 다음 block
    void *prev = (void *)READ_WORD(GET_PREV_FREE_PTR(ptr));  // 이전 block

    if (next != NULL)
    {
        WRITE_WORD(GET_PREV_FREE_PTR(next), (unsigned int)prev); // 다음 block의 이전 포인터 설정
    }

    if (prev != NULL)
    {
        WRITE_WORD(GET_NEXT_FREE_PTR(prev), (unsigned int)next); // 이전 block의 다음 포인터 설정
    }
    else
    {
        segregated_free_lists[index] = next; // free 목록의 인덱스 설정
    }
}
```

  - delete_free_block: free block을 segregated free list에서 삭제한다.
  - 주어진 free block을 segregated list에서 제거한다. block의 위치에 따라 포인터를 업데이트하여 list에서 제거한다.

-
```c
/* 적절한 목록 인덱스를 결정 */
static int find_list_index(size_t size)
{
    int index = 0;                                  // 인덱스 초기화
    while (size > 1 && index < NUM_FREE_LISTS - 1) // 크기가 1보다 크고 인덱스가 NUM_FREE_LISTS-1보다 작은 동안 반복
    {
        size >>= 1; // 크기를 오른쪽으로 1 비트 시프트
        index++;    // 인덱스 증가
    }
    return index; // 인덱스 반환
}
```

  - find_list_index: 주어진 크기에 해당하는 segregated free list의 인덱스를 찾는다.
  - block의 크기에 따라 적절한 list 인덱스를 결정한다. block 크기를 오른쪽으로 비트 시프트하여 적절한 인덱스를 찾는다.

- insert_free_block 함수는 새로운 free block을 segregated list에 삽입하는 기능을 한다. delete_free_block 함수는 연결 list에서 주어진 block을 제거하는 기능을 한다. find_list_index 함수는 block 크기에 맞는 list 인덱스를 찾는다.

- Block 병합 및 배치

-
```c
/* block 병합 및 배치 */
static void coalesce_with_prev(void *ptr, size_t *size, void **new_ptr); // 이전 block과 합치는 함수
static void coalesce_with_next(void *ptr, size_t *size);                 // 다음 block과 합치는 함수
static void *coalesce(void *ptr);                                        // block을 병합하는 함수
static void *place(void *ptr, size_t asize);                             // block을 배치하는 함수
```

-
```c
/* 인접한 free block 병합 */
static void *coalesce(void *ptr)
{
    size_t size = READ_SIZE(GET_HEADER(ptr));                    // 현재 block의 크기
    int prev_alloc = READ_ALLOC(GET_HEADER(GET_PREV_BLOCK(ptr))); // 이전 block의 할당 여부
    int next_alloc = READ_ALLOC(GET_HEADER(GET_NEXT_BLOCK(ptr))); // 다음 block의 할당 여부

    if (READ_TAG(GET_HEADER(GET_PREV_BLOCK(ptr)))) // 이전 block에 태그가 있는지 확인
        prev_alloc = 1;                            // 태그가 있으면 할당된 것으로 간주

    if (prev_alloc == 1 && next_alloc == 1) // 이전과 다음 block 모두 할당된 경우
    {
        return ptr; // 현재 block 반환
    }

    void *new_ptr = ptr; // 새로운 block 포인터 초기화

    if (prev_alloc == 1 && !next_alloc) // 이전 block이 할당되고 다음 block이 free인 경우
    {
        coalesce_with_next(ptr, &size); // 다음 block과 병합
    }
    else if (!prev_alloc && next_alloc == 1) // 이전 block이 free이고 다음 block이 할당된 경우
    {
        coalesce_with_prev(ptr, &size, &new_ptr); // 이전 block과 병합
    }
    else if (!prev_alloc && !next_alloc) // 이전과 다음 block 모두 free인 경우
    {
        delete_free_block(ptr);                                                                          // 현재 block 삭제
        delete_free_block(GET_PREV_BLOCK(ptr));                                                          // 이전 block 삭제
        delete_free_block(GET_NEXT_BLOCK(ptr));                                                          // 다음 block 삭제
        size += READ_SIZE(GET_HEADER(GET_PREV_BLOCK(ptr))) + READ_SIZE(GET_HEADER(GET_NEXT_BLOCK(ptr))); // 크기 업데이트
        new_ptr = GET_PREV_BLOCK(ptr);                                                                   // 새로운 block 포인터 업데이트
        WRITE_WORD(GET_HEADER(new_ptr), PACK_SIZE_AND_ALLOC(size, 0));                                   // header 업데이트
        WRITE_WORD(GET_FOOTER(new_ptr), PACK_SIZE_AND_ALLOC(size, 0));                                   // footer 업데이트
    }

    insert_free_block(new_ptr, size); // 새로운 block 삽입
    return new_ptr;                   // 새로운 block 포인터 반환
}
```

- coalesce: 인접한 free block을 병합하여 메모리 단편화를 줄인다.
- 이전 및 다음 block이 free block인지 확인하고, 병합한 후 새로운 block을 free list에 삽입한다. 병합할 수 있는 경우 block을 삭제하고, 크기를 업데이트한 후 다시 삽입한다.

```
// free block을 확장하고 분리된 free 목록에 입력
static void coalesce_with_next(void *ptr, size_t *size)
{
    if (ptr == NULL || size == NULL)
    {
        fprintf(stderr, "Error: Invalid pointer or size in coalesce_with_next.\n"); // 오류 메시지 출력
        return;                                                                     // 함수 종료
    }

    void *next_ptr = GET_NEXT_BLOCK(ptr); // 다음 block 포인터
    if (next_ptr == NULL)
    {
        fprintf(stderr, "Error: Next block is null in coalesce_with_next.\n"); // 오류 메시지 출력
        return;                                                                // 함수 종료
    }

    delete_free_block(ptr);                                    // 현재 block 삭제
    delete_free_block(next_ptr);                               // 다음 block 삭제
    *size += READ_SIZE(GET_HEADER(next_ptr));                  // 크기 업데이트
    WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(*size, 0)); // header 업데이트
    WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(*size, 0)); // footer 업데이트
}
```

- coalesce_with_next: 현재 block과 다음 block을 병합하여 free block을 확장하고, 이를 segregated free list에 삽입한다.
- ptr 또는 size가 NULL인 경우 오류 메시지를 출력하고 함수 실행을 종료한다.
- next_ptr는 ptr 다음 block의 포인터를 가리킨다. 만약 next_ptr가 NULL인 경우 오류 메시지를 출력하고 함수 실행을 종료한다.
- 현재 block과 다음 block을 free list에서 삭제한다.
- 현재 block의 크기에 다음 block의 크기를 더한다.
- 병합된 block의 header와 footer를 업데이트하여 새로운 크기를 설정한다.

```
static void coalesce_with_prev(void *ptr, size_t *size, void **new_ptr)
{
    if (ptr == NULL || size == NULL || new_ptr == NULL)
    {
        fprintf(stderr, "Error: Invalid pointer or size in coalesce_with_prev.\n"); // 오류 메시지 출력
        return;                                                                     // 함수 종료
    }

    void *prev_ptr = GET_PREV_BLOCK(ptr); // 이전 block 포인터
    if (prev_ptr == NULL)
    {
        fprintf(stderr, "Error: Previous block is null in coalesce_with_prev.\n"); // 오류 메시지 출력
        return;                                                                    // 함수 종료
    }

    delete_free_block(ptr);                                    // 현재 block 삭제
    delete_free_block(prev_ptr);                               // 이전 block 삭제
    *size += READ_SIZE(GET_HEADER(prev_ptr));                  // 크기 업데이트
    *new_ptr = prev_ptr;                                       // 새로운 block 포인터 업데이트
    WRITE_WORD(GET_HEADER(*new_ptr), PACK_SIZE_AND_ALLOC(*size, 0)); // header 업데이트
    WRITE_WORD(GET_FOOTER(*new_ptr), PACK_SIZE_AND_ALLOC(*size, 0)); // footer 업데이트
}
```

- coalesce_with_prev: 현재 block과 이전 block을 병합하여 free block을 확장하

고, 이를 segregated free list에 삽입한다.

- ptr, size 또는 new_ptr가 NULL인 경우 오류 메시지를 출력하고 함수 실행을 종료한다.
- prev_ptr는 ptr 이전 block의 포인터를 가리킨다. 만약 prev_ptr가 NULL인 경우 오류 메시지를 출력하고 함수 실행을 종료한다.
- 현재 block과 이전 block을 free list에서 삭제한다.
- 현재 block의 크기에 이전 block의 크기를 더한다.
- 병합된 block의 시작 포인터를 이전 block으로 설정한다.
- 병합된 block의 header와 footer를 업데이트하여 새로운 크기를 설정한다.

```c
/* block을 free 목록에 배치 */
static void *place(void *ptr, size_t asize)
{
    size_t size = READ_SIZE(GET_HEADER(ptr)); // 현재 block의 크기
    size_t free_size = size - asize;          // free block의 크기

    delete_free_block(ptr); // 현재 block 삭제

    if (free_size < 2 * DWORD_SIZE) // free block의 크기가 최소 크기보다 작은 경우
    {
        WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(size, 1)); // header 업데이트
        WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(size, 1)); // footer 업데이트
    }
    else
    {
        WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(asize, 1)); // header 업데이트
        WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(asize, 1)); // footer 업데이트

        void *next_block = GET_NEXT_BLOCK(ptr);                                 // 다음 block 포인터
        WRITE_WORD(GET_HEADER(next_block), PACK_SIZE_AND_ALLOC(free_size, 0)); // 다음 block header 업데이트
        WRITE_WORD(GET_FOOTER(next_block), PACK_SIZE_AND_ALLOC(free_size, 0)); // 다음 block footer 업데이트
        insert_free_block(next_block, free_size);                               // free block 삽입
    }
    return ptr; // block 포인터 반환
}
```

- place: 요청된 크기의 메모리 block을 free block에 배치하고, 필요 시 block을 분할하여 남은 부분을 free block으로 유지한다.
- 할당된 block을 free block에 배치하고, 필요한 경우 block을 분할한다. 남은 block이 충분히 크다면 free block으로 분할하고, 그렇지 않으면 전체 block을 할당한다.

- coalesce 함수는 인접한 free block을 병합하여 메모리 단편화를 줄인다. place 함수는 할당된 block을 free block에 배치하고, 필요한 경우 block을 분할한다.
- Heap 확장 및 메모리 할당/free

```
/* heap 확장 및 메모리 할당/free */
static void *extend_heap(size_t size); // heap을 확장하는 함수
void mm_free(void *ptr);
void *mm_malloc(size_t size);
void *mm_realloc(void *ptr, size_t size);
```

```
/* heap을 새로운 free block으로 확장 */
static void *extend_heap(size_t size)
{
    void *ptr = mem_sbrk(size); // 메모리 할당 요청
    if (ptr == (void *)-1)      // 할당 실패 시
    {
        fprintf(stderr, "Error: Could not extend heap.\n"); // 오류 메시지 출력
        return NULL;                                        // NULL 반환
    }

    WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(size, 0));              // header 업데이트
    WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(size, 0));              // footer 업데이트
    WRITE_WORD(GET_HEADER(GET_NEXT_BLOCK(ptr)), PACK_SIZE_AND_ALLOC(0, 1)); // Epilogue header 업데이트

    insert_free_block(ptr, size); // free block 삽입
    return coalesce(ptr);         // 병합 후 반환
}
```

- extend_heap: 힙의 크기를 늘린다.
- 메모리를 요청하여 힙을 확장하고, 새로운 free block을 초기화한 후 free list에 삽입한다. 힙 확장에 실패하면 NULL을 반환한다.

```
/* block 해제 */
void mm_free(void *ptr)
{
    if (ptr == NULL) // 포인터가 NULL이면
    {
        return; // 함수 종료
    }
    size_t size = READ_SIZE(GET_HEADER(ptr));                  // block의 크기
    WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(size, 0)); // header 업데이트
    WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(size, 0)); // footer 업데이트
    insert_free_block(ptr, size);                              // free block 삽입
    coalesce(ptr);                                             // 병합
}
```

- mm_free: block을 해제한다.
- block을 해제하고, 인접한 free block과 병합한 후 free list에 삽입한다. 해제된 block의 header와 footer를 업데이트한다.

```c
/* block 할당 */
void *mm_malloc(size_t size)
{
    if (size == 0)   // 크기가 0이면
        return NULL; // NULL 반환

    size_t asize;       // 조정된 크기
    size_t extend_size; // 확장 크기
    void *ptr;          // block 포인터

    if (size <= DWORD_SIZE)     // 크기가 DWORD_SIZE 이하이면
        asize = 2 * DWORD_SIZE; // 조정된 크기 설정
    else
        asize = DWORD_SIZE * ((size + (DWORD_SIZE) + (DWORD_SIZE - 1)) / DWORD_SIZE); // 조정된 크기 계산

    ptr = find_fit(asize); // 적절한 block 찾기

    if (ptr == NULL) // 적절한 block이 없으면
    {
        extend_size = MAX_VALUE(asize, INITIAL_CHUNK_SIZE); // 확장 크기 설정
        ptr = extend_heap(extend_size);                     // heap 확장
        if (ptr == (void *)-1)                              // heap 확장 실패 시
            return NULL;                                    // NULL 반환
    }
    ptr = place(ptr, asize); // block 배치
    return ptr;              // block 포인터 반환
}
```

- mm_malloc: 메모리 block을 할당한다.
- 요청된 크기를 조정하고, 적절한 free block을 찾아 배치하며, 포인터를 반환한다. 적절한 block을 찾지 못한 경우 힙을 확장한다.

```c
/* block 재할당 */
void *mm_realloc(void *ptr, size_t size)
{
    if (size == 0) // 크기가 0이면
    {
        mm_free(ptr); // block 해제
        return NULL;  // NULL 반환
    }

    if (ptr == NULL) // 포인터가 NULL이면
    {
        return mm_malloc(size); // 새 block 할당
    }

    size_t asize; // 조정된 크기

    // 조정된 크기 계산
    if (size <= DWORD_SIZE)
        asize = 2 * DWORD_SIZE;
    else
        asize = DWORD_SIZE * ((size + (DWORD_SIZE) + (DWORD_SIZE - 1)) / DWORD_SIZE);

    // 현재 block이 충분히 큰 경우
    if (READ_SIZE(GET_HEADER(ptr)) >= asize)
        return ptr;

    void *next_block = GET_NEXT_BLOCK(ptr);           // 다음 block 포인터
    size_t current_size = READ_SIZE(GET_HEADER(ptr)); // 현재 block 크기

    // 다음 block이 free block인 경우
    if (!READ_ALLOC(GET_HEADER(next_block)) || !READ_SIZE(GET_HEADER(next_block)))
    {
        size_t next_size = READ_SIZE(GET_HEADER(next_block)); // 다음 block 크기
        size_t total_size = current_size + next_size;         // 총 크기

        if (total_size >= asize) // 총 크기가 조정된 크기보다 크거나 같은 경우
        {
            delete_free_block(next_block); // 다음 block 삭제

            // 새로운 크기로 header와 footer 업데이트
            WRITE_WORD(GET_HEADER(ptr), PACK_SIZE_AND_ALLOC(total_size, 1));
            WRITE_WORD(GET_FOOTER(ptr), PACK_SIZE_AND_ALLOC(total_size, 1));

            return ptr; // block 포인터 반환
        }
    }

    // 현재 block과 이웃 block이 충분하지 않은 경우 새 block 할당
    void *newptr = mm_malloc(asize); // 새 block 할당
    if (newptr == NULL)              // 할당 실패 시
    {
        fprintf(stderr, "Error: Could not allocate memory for realloc.\n"); // 오류 메시지 출력
        return NULL;                                                        // NULL 반환
    }

    // 데이터를 새 block으로 복사
    memcpy(newptr, ptr, MIN_VALUE(size, asize)); // 데이터 복사
    mm_free(ptr);                                // 기존 block 해제

    return newptr; // 새 block 포인터 반환
}
```

    - mm_realloc: 메모리 block을 재할당한다.
    - 요청된 크기를 조정하고, 현재 block을 확장할 수 있는지 확인한다. 확장이 불가능하면 새로운 block을 할당하고 데이터를 복사한 후 이전 block을 해제한다.

- extend_heap 함수는 힙의 크기를 늘린다. mm_free 함수는 block을 해제하고, mm_malloc 함수는 메모리 block을 할당한다. mm_realloc 함수는 메모리 block을 재할당한다. 각 함수는 block을 free list에 추가하고, 필요한 경우 병합한다.